Learning Sequential and Parallel Runtime Distributions for Randomized Algorithms

Alejandro Arbelaez
Insight Centre for Data Analytics
University College Cork, Ireland
alejandro.arbelaez@insight-centre.org

Charlotte Truchet
LINA, UMR 6241
University of Nantes, France
charlotte.truchet@univ-nantes.fr

Barry O'Sullivan
Insight Centre for Data Analytics
University College Cork, Ireland
barry.osullivan@insight-centre.org

***Abstract*—In cloud systems, computation time can be rented by the hour and for a given number of processors. Thus, accurate predictions of the behaviour of both sequential and parallel algorithms has become an important issue, in particular in the case of costly methods such as randomized combinatorial optimization tools. In this work, our objective is to use machine learning algorithms to predict performance of sequential and parallel local search algorithms. In addition to classical features of the instances used by other machine learning tools, we consider data on the sequential runtime distributions of a local search method. This allows us to predict with a high accuracy the parallel computation time of a large class of instances, by learning the behaviour of the sequential version of the algorithm on a small number of instances. Experiments with three solvers on SAT and TSP instances indicate that our method works well, with a correlation coefficient of up to 0.85 for SAT instances and up to 0.95 for TSP instances.**

I. Introduction

For decades, solving a difficult SAT instance has been a challenge. Now, with the improvements of the solvers and the availability of computation time in parallel architectures, that can be rented by the hour, the problem has changed. The question, as in [1] for instance, is no longer "how much time would I need to solve an instance", but rather "since I have paid for x hours of computation on n cores, how many SAT instances could I solve?". In this paper, we propose a method for solving large classes of SAT instances with randomized algorithms, by learning features related to the solving time on a small set of instances and applying this knowledge to predict the solving time for the other instances.

The originality of our method, and what makes its prediction accurate, lies in the critical parameters we learn about the runtime distribution (RTD) of the target algorithm. Unlike traditional methods that learn single metrics such as mean or median runtime of the algorithm, we learn the actual runtime distribution. Therefore, we have the complete knowledge of the behaviour of the sequential algorithm, from which the runtime of its multiwalk parallel version can be inferred. To do this, we use the probabilistic model proposed by [2] for approximating the solving time of multiwalk parallel local search algorithms.

The runtime distribution can be learned by running the algorithm several times on a particular instance, which we do on a small set of instances (the learning set). On this set, we can approximate the type of distribution, its expectation and its variance. Once this is done, we use pace regression [3] to predict the critical parameters of the distribution for the unsolved instances, hence to predict the resolution time (sequential and parallel). We expect this work to have potential applications in multiple areas such as: algorithm selection, parameter tuning, and modelling the behaviour of the algorithms.

The remainder of the paper is organized as follows. Section II presents background material on machine learning, runtime distributions, and our case study problem domains, i.e., SAT and TSPs. Section III details the proposed methodology. Section IV reports on the experimental validation. Section V discusses related work with a focus on the current methodologies for estimating the performance of a given algorithm. Finally, Section VI concludes the paper and dicusses directions for future work.

II. Background

A. Problem Domains

The *Boolean Satisfiability problem* (SAT) involves determining whether a given Boolean formula F is satisfiable or not. This formula is usually represented using the *Conjunctive Normal Form* (CNF) as follows: $F = \bigwedge_i \bigvee_j l_{ij}$, where each l_{ij} represents a literal (a propositional variable or its negation) and the disjunctions $\bigvee_j l_{ij}$ are the clauses in F.

Solving F involves deciding whether or not there exists a model (so-called solution) in which the truth assignment for the variables in F satisfies all clauses, or demonstrating that no such assignment exists. Currently, there are two well established techniques for solving SAT formulas; complete and incomplete search. State-of-the-art complete solvers, e.g. [4]–[6], are developed based on the DPLL algorithm. They combine a tree-based search with unit propagation, conflict-clause learning, and intelligent backtracking. State-of-the-art incomplete solvers are mainly based on local search algorithms [7] to find a truth assignment for the variables which satisfies all the clauses. Alternatively, hybrid algorithms [8], [9] combine local search with clause learning to devise a complete algorithm.

The *Traveling Salesman Problem* (TSP) is a well-known combinatorial optimisation problem with applications in multiple areas ranging from transportation to VLSI circuit design, and bioinformatics. The TSP consists in finding the shortest path, or tour, among a predefined list of n cities such that all cities are visited only once. In this paper we focus our attention in the 2D Euclidean TSP in which we are given a set S of points in a plane, and the distance between two points is the Euclidean distance between their corresponding coordinates.

Local search algorithms for the TSP typically use the k-opt operator [10]. The operator iteratively selects k edges from the current solution and replaces them with k new edges by improving the objective function. Alternatively, complete solvers (e.g., Concorde [11]) combine the use of branch-and-bound with cutting algorithms to find optimal tours.

In the remainder of this paper we consider the performance of local search algorithms on these two problem domains. For this purpose, we need high-level data that describes the instances, which we will describe below.

B. Problem Features

1) SAT Features: We will use a widely known set of problem descriptors or features for SAT previously described in [12]. In particular, we use the feature computation code available from the Beta lab at the University of British Columbia.[1] This tool can compute several collections of features describing the SAT instances. Some are static, such as: number of variables, clauses, ratio variables/clauses, etc. It also collects relevant information of two local search algorithms, namely SAPS and GSAT, to compute information of the trajectory of the LS process such as: number of steps to the best local minimum in a run (mean, median, 10th and 90th percentiles) for SAPS; average improvement to best run: mean improvement per step to best solution for SAPS; fraction of improvement due to first local minimum: mean for SAPS and GSAT; coefficient of variation of the number of unsatisfied clauses in each local minimum: mean over all runs for SAPS. The tool can use Knuth's sampling method [13] to estimate the number of nodes and the depth of the search tree. In total we will use 81 of these features. The same reference also describes a set of features involving the use of DPLL and MIP based solvers for a short time, however, these features appear to be irrelevant in our case, since we use local search algorithms.

2) TSP Features: In this paper we use the set of features for TSP problem described in [14]. In particular, we use the feature computation code, again, from the Beta lab at the University of British Columbia.[2] We use a set of 50 features, collecting general statistics of the target problem instance such as number of cities; cost matrix features; statistics on the resulting minimum spanning tree of the input instance; relevant information of the local search trajectory for the LKH solver [10]; and branch and cut features collecting statistics of the concorde solver.

C. Runtime Distributions of Las Vegas Algorithms

Local Search is a well-known technique to solve combinatorial problems in multiple domains. Generally speaking, a local search algorithm starts with a random initial solution or assignment to the variables, and iteratively improves the solution by performing small changes until a stopping criterion is met. Local Search algorithms include several random components: choice of an initial configuration, choice of a move among several candidates, plateau mechanism, random restart, etc, and their runtime varies from one execution to another even on the same input. Indeed, these algorithms can be considered into the larger framework of *Las Vegas Algorithms* [15], i.e., randomized algorithms whose runtime might vary from one execution to another, even with the same input instance.

The multi-walk method [16] is a popular parallel local search algorithm that does not require to modify the base algorithm. It consists in executing multiple copies of a given algorithm with different random seeds until a solution is obtained. Ideally, one would like a linear speed-up, that is, the runtime required to solve a given instance decreases proportionally with the number of processing units.

Typically, the most popular metric to evaluate the performance of sequential and parallel local search algorithms is the average runtime. However, a more detailed analysis could be obtained by studying the execution time of the algorithm as a random variable and performing a statistical analysis of its probability function. This analysis leads to the runtime distribution (RTD), a powerful tool that describes the probability of a given algorithm to find a solution for a given instance within a given amount of time.

The RTD is heavily used to analyze the behaviour of randomised algorithms. In [7], the authors use RTD to finely tune local search algorithms, additionally [17] uses the runtime distribution to define optimal restart strategies in sequential and parallel algorithms and to provide bounds on the parallel expectation.

Recently, [2] proposed a methodology to estimate the runtime distribution of the parallel algorithm with a statistical analysis, based on order statistics, of the sequential version. The methodology assumes that the multi-walk algorithm (sometimes called parallel portfolio) executes multiple copies of the algorithm, with different random seeds. Thus, the processes are independent, and identically distributed.

Roughly speaking, the runtime distribution describes the performance of the algorithm and provides the probability density function of a given algorithm to solve a given instance within some time. Let $f_Y(x)$ and $\mathcal{F}_Y(x)$ be the probability distribution and cumulative distribution functions of the sequential algorithm. The probability functions give return the probability that the runtime of the algorithm is smaller that a given time x. Let $Z^{(n)}$ be a random variable denoting the runtime of a given parallel algorithm implementing the multi-walk framework with n cores to solve a given instance within some time, with probability distribution $f_{Z^{(n)}}(x)$, and the cumulative distribution $\mathcal{F}_{Z^{(n)}}(x)$ that can be computed as follows:

$$\begin{aligned}
\mathcal{F}_{Z^{(n)}}(x) &= \mathbb{P}[Z^{(n)} \leq x] \\
&= \mathbb{P}[\exists i \in \{1...n\}, X_i \leq x] \\
&= 1 - \mathbb{P}[\forall i \in \{1...n\}, X_i > x] \\
&= 1 - \textstyle\prod_{i=1}^{n} \mathbb{P}[X_i > x] \\
&= 1 - (1 - \mathcal{F}_Y(x))^n
\end{aligned}$$

The derivative of the cumulative probability function leads to the probability density function of the parallel algorithm as

[1] Code available at www.cs.ubc.ca/labs/beta/Projects/SATzilla. We use the following settings: *-base*, *-ls*, and *-lobjois*

[2] Code available at http://www.cs.ubc.ca/labs/beta/Projects/EPMs, we use the following setting: *-all*

follows:

$$\begin{aligned} f_{Z^{(n)}} &= (1-(1-\mathcal{F}_Y)^n)' \\ &= n f_Y (1-\mathcal{F}_Y)^{n-1} \end{aligned}$$

The expected runtime of the parallel algorithm (assuming the multi-walk framework) is defined as follows:

$$\begin{aligned} \mathbb{E}\left[Z^{(n)}\right] &= \int_0^\infty t f_{Z^{(n)}}(t)\ dt \\ &= n\int_0^\infty t f_Y(t)(1-\mathcal{F}_Y(t))^{n-1}\ dt \end{aligned}$$

In [2] the authors show that the RTD for different local search algorithms and a variety of instances can be characterised using two types of distributions: shifted exponential and lognormal. An algorithm is exponentially distributed if the probability of finding a solution within time t is given by:

$$\mathcal{F}_Y(t) = \begin{cases} 0 & \text{if } t \leq x_0 \\ 1-e^{-\lambda(t-x_0)} & \text{if } t > x_0 \end{cases}$$

where the parameter x_0 represents the shift of the distribution. An important property of an algorithm *exponentially distributed* is that its expectation when the number of cores tends to infinity is 0. However, as shown in [18] the speedup for the shifted exponential distribution is limited to $1+\frac{1}{x_0\ \lambda}$, where x_0 is the minimum value of the distribution and λ = 1/mean. Intuitively, a shifted distribution means that a minimum number iterations are needed to find a solution.

Alternatively, an algorithm is *lognormally distributed* if the probability of finding a solution within time t is given by:

$$\mathcal{F}_Y(t) = \begin{cases} 0 & \text{if } t \leq 0 \\ \frac{e^{-(\ln t-\mu)^2/(2\cdot\sigma^2)}}{t\cdot\sigma\cdot\sqrt{2\pi}} & \text{if } t > 0 \end{cases}$$

From these formulas, the expected runtime of the multiwalk parallel algorithm can be derived either symbolically (in case of the exponential) or numerically (in case of the lognormal). The key parameters of shifted exponentially distributed algorithms are the shift x_0 and λ denoting 1 divided the mean runtime of the algorithm. The key parameters for a lognormally distributed algorithm are μ and the standard deviation σ of the natural logarithm of the runtime.

The lognormal distribution is a typical example of a fat-tailed distribution [19], and this distribution is known to be a good characterisation for local search algorithms. Unlike the exponential distribution where it is feasible to analytically solve the integral under the curve to compute the expected value, for the lognormal distribution it is necessary to use numerical methods to compute the expected value.

We remark that [2] computes the RTD of the algorithms after collecting a representative number of runtime executions of the algorithms. Informally speaking, [2] requires that an instance is solved in order to analyse its RTD. In this paper, we overcome this limitation by using ML techniques to predict the parameters of the distributions of unseen instances.

D. Supervised Machine Learning

Supervised Machine Learning (SML) exploits data labelled by an expert to automatically build hypotheses emulating the expert's decisions [20]. Formally, a learning algorithm works on a space of features Ω which are usually vectors of d values in a given set R, *i.e.* $\Omega = R^d$. To each vector of features is assigned an output which can be numerical or a class label (in the case of classification). Let Y be the set of outputs; in case of multiple outputs Y can be a Cartesian product of numerical or discrete sets. An SML algorithm first processes a training set $\mathcal{E} = \{(x_i, y_i), x_i \in \Omega, y_i \in Y, i = 1 \ldots k\}$ made of k examples (x_i, y_i), where $x_i \in \Omega$ is the example description, and y_i is the associated output. The learning algorithm outputs a hypothesis $f : \Omega \mapsto Y$ associating to each example description x the output $y = f(x)$. The function f is built from the training set and can then be used to predict the output for other examples.

For the case of numerical outputs, linear regression is among the most prominent regression techniques. This technique considers real-valued features ($\Omega = \mathbb{R}^d$) and constructs a function $f(x) = \beta_0 + \beta_1 x$ that best fits the training data. *Pace regression* is a linear regression technique to compute β_0 and β_1 with outstanding performance in particular when the number of features is large and not completely mutually independent.

In the binary classification case, the output of the learning algorithm is either positive or negative ($Y = \{-1, 1\}$). Among the most prominent classification algorithms are support vector machines. This classification algorithm constructs the separating hyperplane that maximises the margin, i.e., the minimal distance between the examples and the separating hyperplane. The margin maximisation principle provides good guarantees about the stability of the solution.

III. The Approach

We describe here our method, which combines machine learning and statistical evaluation to predict the computation time of a randomized algorithm for large classes of SAT instances. In the following, we assume that we have:

- a class of instances of a given problem, assumed to be similar (for instance, k-SAT instances for a constant k),
- a local search algorithm $\mathcal{A}$ for the problem.

For sake of clarity, we describe the method in the case of the SAT problem.

A. Offline Learning Phase

The first step consists in defining the training set and compute its features and outputs. For this, we proceed in two phases. In the first phase, we compute the set of SAT features R_1 as defined in SATzilla. Most of them are static, and we keep the features concerning the solving time for SAPS and GSAT as described above in Section II-B2. In the second phase, we run $\mathcal{A}$ a certain number p of times, and we collect all these runtimes. From this, we approximate the distribution of

the runtime of $\mathcal{A}$ on the training instances using the method described in Section II-C. From this, we collect three data points: the type of distribution of the RTD, its expectation and its variance. These three data are the output that we are interested in.

Concerning the type of distribution, we make the assumption, that is does not change inside a given class of problem. Precisely, if all 3-SAT instances for a particular clause-to-variable ration in the training set correspond to a lognormal distribution, we will take the lognormal distribution for every new 3-SAT instance. For the parameters, we need a numerical estimation which is done with a linear regression method. In Section IV-A we describe how we select the most suitable distribution for the target problems.

In case of a lognormal distribution, we collect the expectations and variances of all the training set: $\mathcal{E}_\mu = \{(x_1, \mu_1), \ldots, (x_k, \mu_k)\}$ and $\mathcal{E}_\sigma = \{(x_1, \sigma_1), \ldots, (x_k, \sigma_k)\}$. In case of a shifted exponential distribution, we collect the minimum and expectation of all the training set: $\mathcal{E}_m = \{(x_1, m_1), \ldots (x_k, m_k)\}$ and $\mathcal{E}_\lambda = \{(x_1, \lambda_1), \ldots, (x_k, \lambda_k)\}$. Then we learn the functions allowing us to estimate the parameters (one function for each parameter), for instance in case of a lognormal distribution we learn a function estimating the expectation and another function estimating the variance: $f_\sigma(x)$=β_0^σ + $\beta_1^\sigma x$ and $f_\mu(x)$=β_0^μ, $\beta_1^\mu x$. This learning phase is used to calibrate the model, and it happens offline.

B. Predicting the Sequential and Parallel Runtime Distribution

For unsolved instances we use our regression models to estimate the runtime distribution of the sequential distribution. First, we collect the vector of features characterising the new instance x_i in a given class. Then we use f_σ and f_μ if the best fit was the lognormal distribution for this class of instances during the learning phase, and f_m and f_λ for the shifted exponential distribution. Finally, we feed the cumulative distribution function with the output of the regression models to predict the performance of the algorithm, using the model defined in Section II-C.

With the same notation as we used in Section II-C, the mean of the cumulative distributions for the sequential algorithms can be calculated symbolically as $x_0 + 1/\lambda$ for the shifted exponential distribution, and as $e^{u+\sigma^2/2}$ the lognormal distribution. For the parallel algorithms we use the expectation, i.e., $\mathbb{E}\left[Z^{(n)}\right]$, where n indicates the number of cores. The integral can be solved symbolically for the shifted exponential distribution to compute the mean runtime as $min + 1/(n \cdot \lambda)$. For the lognormal distribution we use Mathematica to compute the integral.

IV. Experimental Evaluation

We focus our attention on three SAT local search solvers (Sparrow [21], probSAT [22], and g2wsat [23]), and a TSP local search solver LKH [10]. We recall that these solvers are among the best solvers in their particular domains and we use WEKA (version 3.6.2), a popular ML library to build the ML models. In particular, we use the *Pace regression* algorithm available in WEKA with its default parameters for the machine learning models to estimate the parameters of the distributions, i.e., lognormal and shifted exponential.

In our experiments we use random and structured instances. The random SAT set of instances has been used previously in [24] and [22] and we use the software *portgen* to randomly generate TSP instances. The set of structured SAT instances has been proposed in [25] and previously in [26], and we collected structured TSP instances from OpenStreetMaps (osm).[3]

Random instances:

- 500 $\times$ 3-SAT instances with 10,000 variables around the phase transition with 42,000 clauses and a clause-to-variable ratio (r) of 4.2;
- 500 $\times$ 5-SAT instances with 500 variables, 10000 clauses and r=20;
- 500 $\times$ TSP instances with the number of cities varying from 500 to 5000.

Structured instances:

- 500 $\times$ SAT-encoded small-world graph colouring problems (sw);
- 74 $\times$ TSP-encoded cities from OpenStreetMaps, each instance contains a set of points of interests of the city.

Sparrow and probSAT are known to be very effective for random instances and [26] showed that g2wsat is a robust alternative for sw instances. In total we evaluate five scenarios for random instances ($\langle$3-SAT, Sparrow$\rangle$, $\langle$3-SAT, probSAT$\rangle$, $\langle$5-SAT, Sparrow$\rangle$, $\langle$5-SAT, probSAT$\rangle$, and $\langle$TSP-random, LKH$\rangle$) and two scenarios for structured instances ($\langle$SAT-sw, g2wsat$\rangle$ and $\langle$TSP-osm, LKH$\rangle$). We treat each scenario independently, therefore, we learn and test an algorithm for each reference problem class.

We use 10-fold cross-validation to evaluate the accuracy of our machine learning approach. In 10-fold cross-validation the entire data set is divided into 10 disjoint sets $D = \{D_1, D_2, \ldots, D_{10}\}$. For each subset $D_{i\in[1,10]}$, the machine learning model is learned with $D - D_i$ and tested on D_i. The advantage of the 10-fold cross-validation is that all instances in the same problem family are used for both training and testing, but each instance is used for testing exactly once.

We performed all our experiments on a 39-node cluster with Intel Xeon E5430 processors at 2.66Ghz and 12GB of RAM. In all the experiments we use a time limit of 3600 seconds. Nearly all the experiments completed the execution within this time limit, however, in the exceptional cases where the timeout was reached we use the timeout value.

A. Training Phase

During the training phase we collect 500 observations of the runtime for each instance in the training set (except for osm instances where we collect 300 observations per instance). With this information, the first step in the RTD prediction is to identify a theoretical distribution to characterise the empirical data. Therefore, we use Mathematica [27] to compute the parameters for the two reference distributions, i.e., μ and σ (lognormal distribution) and λ and the min. runtime execution (shifted exponential), for each instance in the training set. Furthermore, we use the Kolmogorov-Smirnov (KS) test to find the distribution that best fits the instances. If the two

[3] https://www.openstreetmap.org

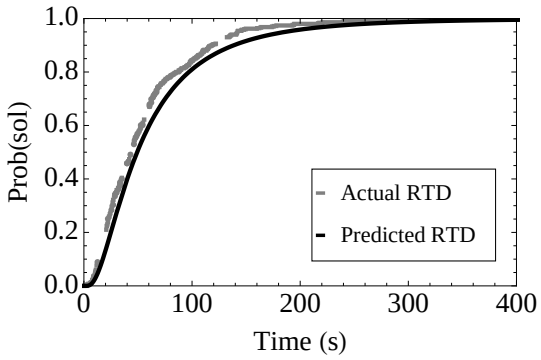


Fig. 1. Actual empirical RTD vs. predicted RTD for sequential probSAT on a typical 3-SAT instance

reference distributions pass the test, we break ties in favor of the distribution with the highest p-value.

In all iterations of the 10-fold cross validation process we observed a single winning distribution for each scenario, that is, the lognormal distribution is the best fit for: ⟨3-SAT, Sparrow⟩, ⟨3-SAT, probSAT⟩, ⟨5-SAT, Sparrow⟩, ⟨TSP-random, LKH⟩, and the shifted exponential distribution is the best fit for: ⟨5-SAT, probSAT⟩. We would like to highlight that we collected statistical evidence that the winning distribution is a good characterisation of the empirical data. For instance, for ⟨3-SAT, probSAT⟩ 389 instances passes the KS test with a p-value better than 5%, and 490 instances passes the test with a p-value better than 1% for the lognormal distribution. For the remaining problem families the winning distribution passes the KS test with a p-value better than 5% for more than 70% of the instances.

Similarly to random instances we also observed a single winner distribution for structured instances, that is, the the non-shifted exponential distribution (i.e., with $x_0 = 0$) is the best fit for ⟨SAT-sw, g2wsat⟩, and the lognormal distribution is the best fit for ⟨TSP-osm, LKH⟩. We also collected enough statistical evidence that the winner distribution is a good characterisation of the empirical observations.

The last step of the training phase consists in computing the regression models for the parameters of the distributions. We experimented with the following regression models from WEKA: Simple linear regression, Linear Regression, Gaussian Process, RBF Network, and Pace Regression. Pace regression reported the overall best results, and therefore we report the results obtained with this method.

B. Testing Phase

In order to evaluate the accuracy of the machine learning models, we perform 10 runs of the multi-walk parallel algorithms and report the mean runtime for each actual empirical evaluation of the algorithms. We recall that for each instance in the testing set, we collect the set of features as described in Section II, and use the regression models to compute the parameters of the best distribution from the learning phase.

We begin our analysis with random instances. To this end, Figure 1 shows the actual vs. predicted RTD for probSAT to solve a typical random 3-SAT instance. Clearly, the predicted RTD closely matches the actual empirical RTD. Similar behaviour has been observed for other instances in the dataset. Certainly, this figure visually confirms the accuracy of the proposed framework to estimate the sequential RTD. Our next step involves inferring the RTD of the parallel algorithm using order statistics as discussed earlier.

TABLE I. ROOT MEAN SQUARE ERROR (RMSE) AND CORRELATION COEFFICIENT (CC) BETWEEN THE PREDICTED AND ACTUAL MEAN RUNTIME FOR RANDOM 3-SAT INSTANCES (WITH 1, 10, 20, 30, AND 40 CORES)

Cores	Sparrow		probSAT	
	RMSE	CC	RMSE	CC
1	0.23	0.82	0.22	0.85
10	0.30	0.78	0.25	0.82
20	0.26	0.79	0.23	0.82
30	0.24	0.82	0.22	0.83
40	0.25	0.79	0.21	0.82

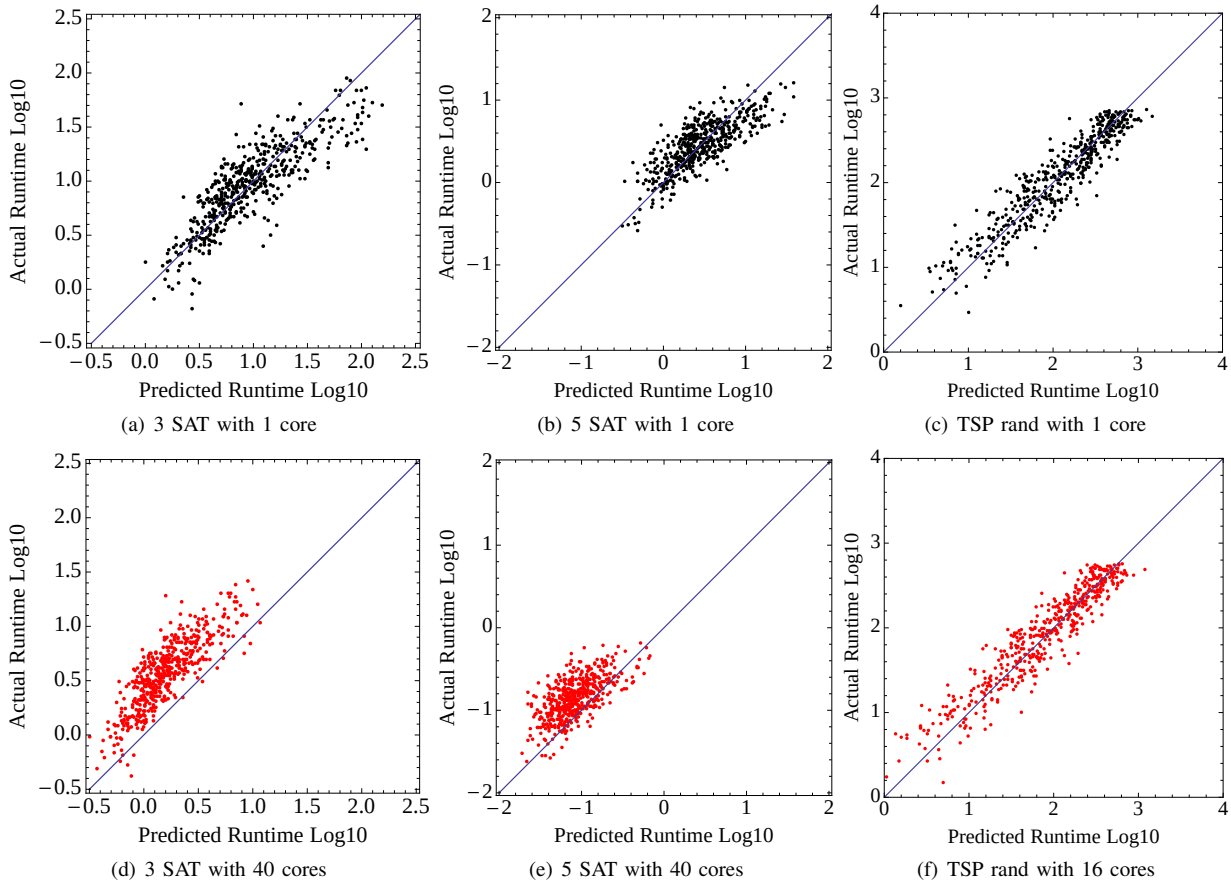
TABLE II. ROOT MEAN SQUARE ERROR (RMSE) AND CORRELATION COEFFICIENT (CC) BETWEEN THE PREDICTED AND ACTUAL MEAN RUNTIME FOR RANDOM 5-SAT INSTANCES (WITH 1, 10, 20, 30, AND 40 CORES)

Cores	Sparrow		probSAT	
	RMSE	CC	RMSE	CC
1	0.24	0.78	0.23	0.79
10	0.49	0.63	0.47	0.60
20	0.43	0.62	0.37	0.60
30	0.38	0.66	0.31	0.65
40	0.35	0.68	0.29	0.66

We now evaluate the quality of the predicted RTD of the parallel algorithms using the multi-walk framework. Figure 2 shows the actual vs. predicted mean runtime for probSAT on 3 and 5 SAT instances with 1, 10, 20, 30, and 40 cores and LKH on TSP instances with 1, 2, 4, 8, and 16 cores. Each point in the figure represents the predicted mean runtime (x-axis) and the actual empirical mean runtime (y-axis), the diagonal line represents a perfect prediction. As expected the sequential algorithm exhibits larger runtimes than the parallel ones, and as we increase the number of parallel units the algorithm tends to report shorter runtimes. It can be visually observed that the predictions greatly match the empirical observations of the sequential and parallel algorithms.

In Tables I, II, and III we consider two of the most common metrics to evaluate the accuracy of machine learning models: the root mean square error (RMSE) and the correlation coefficient (CC) of the predictions vs. the actual runtimes. The RMSE measures the quality of the fit, with 0 representing a perfect prediction. CC measures the direction of the linear relationship between the actual and the predicted runtimes, +1 being a perfect positive fit.

It can be observed that for 3-SAT (resp. 5-SAT) instances the RMSE of the predictions is between 0.21 and 0.25 (resp. 0.24 and 0.49) and the CC ranges from 0.79 to 0.85 (resp. 0.60 to 0.78), and more importantly the accuracy remains stable as we increase the number of cores. The predictions are better for TSP instances with a RMSE close to 0.2 and a CC up to 0.95. We would like to point out that our results for random SAT instances is consistent with the literature for mean runtime predictions, with the benefit that our method estimates the RTD for both sequential and parallel versions of the algorithms. Additionally, we would like to point out that our



(a) 3 SAT with 1 core (b) 5 SAT with 1 core (c) TSP rand with 1 core

(d) 3 SAT with 40 cores (e) 5 SAT with 40 cores (f) TSP rand with 16 cores

Fig. 2. Actual vs. Predicted mean runtime varying the number of cores for probSAT and LKH on SAT and TSP instances

TABLE III. ROOT MEAN SQUARE ERROR (RMSE) AND CORRELATION COEFFICIENT (CC) BETWEEN THE PREDICTED AND ACTUAL MEAN RUNTIME FOR RANDOM TSP INSTANCES (WITH 1, 2, 4, 8, 16 CORES)

	RMSE	CC
1	0.206	0.934
2	0.225	0.936
4	0.213	0.940
8	0.210	0.942
16	0.203	0.950

TABLE IV. ROOT MEAN SQUARE ERROR (RMSE) AND CORRELATION COEFFICIENT (CC) BETWEEN THE PREDICTED AND MEAN RUNTIME FOR STRUCTURED INSTANCES

Cores	g2wsat sw		TSP osm	
	RMSE	CC	RMSE	CC
1	0.702	0.650	0.172	0.882
2	0.836	0.559	0.200	0.884
4	0.636	0.619	0.192	0.891
8	0.431	0.714	0.188	0.892
16	0.256	0.701	0.181	0.893

methodology reports more reliable results than [14] (RMSE of 0.69 vs. 0.2) for the LKH solver on TSP instances with similar characteristics. We attribute this to the fact that [14] uses only one run with a single seed per instance, whereas our method relies on more accurate information by predicting the RTD of the algorithm.

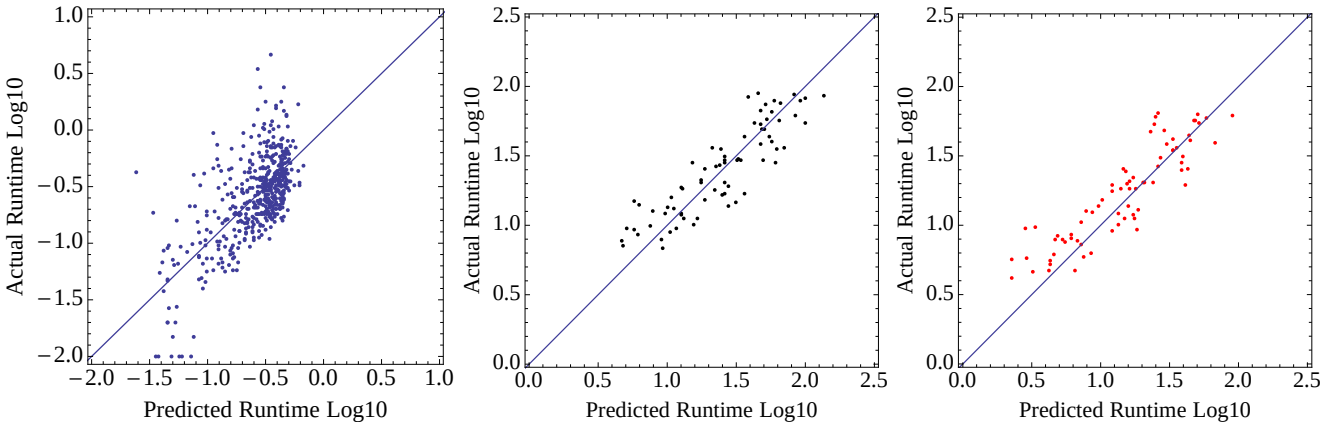
We now move our attention to structured instances. This instances are typically more challenging, however, we recall that we were able to identify a distribution to characterise the two problems. Figure 3 shows the actual vs. predicted mean runtime for g2wsat and LKH on sw and osm instances.

Although the predictions for this set of instances are less accurate than for random instances, we provide reliable estimations for the mean runtime (see Table IV) with a RMSE ranging from 0.25 to 0.7 (resp. 0.88 to 0.89) for g2wat (resp. LKH) and the CC of the predictions is between 0.65 and 0.71 (resp. 0.88 and 0.89) for g2wsat (resp. LKH). We would like to point out that predicting the mean performance on structured instances is more challenging than random instances. The results for g2wsat are consistent with the literature for mean runtime prediction, for instance, [28] showed a CC of 0.6 for SAT local search on structured instances.

Finally, we computed the Spearman's rank correlation coefficient test and observed that in 31 (out of 35) scenarios the reported correlation coefficient is significant with a p-value better than 1%.

V. RELATED WORK

Over the last decades there has been a growing interest in the estimation of the performance of the algorithms to tackle a given instance. Early studies were devoted to estimating the size of the search tree associated with general backtracking algorithms. For example, Knuth's estimator [13] is a Monte Carlo method which randomly explores the tree from the root to the leaves without backtracking, in order to get the expected number of nodes as the average of several executions.



(a) SAT-sw with 1 cores (b) TSP-osm with 1 cores (c) TSP-osm with 16 cores

Fig. 3. Actual vs. Predicted mean runtime varying the number of cores for g2wsat and LKH on structured instances, i.e., sw (SAT) and osm (TSP)

The Knuth's estimator has been extended in several ways. [29] extended the estimator to deal with upper bound solutions which systematically help to prune useless portions of the search tree. [30] proposed an online estimator, that is, estimating the number of remaining nodes from the current state of the search.

Machine learning has also been an alternative to estimate the performance of a large number of algorithms. In [31], the authors propose to use supervised machine learning to classify the variation of the runtime of a given algorithm on a set of Quasigroup instances. The authors use a set of features to train a Bayesian learning algorithm to classify instances into two categories, depending on whether or not their solving time is less or equal than the median time required to solve the entire training set. Afterwards, this information can be used to discard low quality algorithms when solving new instances. Alternatively, [32] shows the use of machine learning to estimate whether a SAT instances is satisfiable or not.

In [14], instead of classifying the runtime of a SAT solver into several categories, addresses the challenging task of predicting its runtime. They collect a set of general features of SAT, TSP, and MIP instances to evaluate the accuracy of several regression methods to estimate the actual runtime of the algorithms. Generally speaking, random forest offers the overall best predictions with a correction coefficient close to 1. [33] combines logistic and linear regression models to make online runtime predictions, that is, predict the remaining time of the algorithm from the current state of the search.

Interestingly, most of the literature for runtime prediction is devoted to deterministic algorithms. However, [28] extended the approach to stochastic local search algorithms algorithms for SAT. To do so, the authors build a linear regression model to predict the mean performance of a given algorithm. Similarly, [34] also deal with the performance of a stochastic algorithm by using neural networks to learn the mean number of iterations for two variations of the LKH solver to tackle TSP instances. In this paper, our approach is close to theirs, but we go beyond predicting the mean or median runtime and predict the RTD of the algorithm, this way, we obtain complete details about the behaviour of the algorithm.

Due to the reliable performance of the machine learning models for runtime prediction, these models have been largely used in the context of the algorithm selection problem to build robust portfolios of algorithms. Informally speaking, a portfolio of algorithms is a framework to select the most suitable algorithm (based on some performance criteria) to solve a given combinatorial problem. Typically, the portfolio learns a machine learning model to predict the runtime of individual algorithms. Classification models have also been employed for the portfolio constructions. We refer the reader to [35] for a recent survey. Our approach is rather orthogonal since we are interested in the performances of a single algorithm.

Recently, [36] demonstrated that analysing the runtime distribution can have important consequences in the outcome of the annual SAT competitions. In particular, the authors showed that due to the variance in the runtime of the solvers, the top tree solvers from the 2014 SAT competition could have been ranked in any permutation by simply re-running the experiments. Additionally, the authors also pointed out that current techniques for estimating the mean runtime of the algorithms might not be enough to provide reliable predictions for industrial SAT instances due to the extreme runtime variation of the solvers.

Finally, [37] proposed the use of case-base reasoning to exploit different parameter configurations in massively parallel systems; [38] evaluates the use parameter tuning approaches in the cloud; and [39] uses machine learning to find a good trade-off between energy consumption and solving time in the context of SAT solving.

VI. Conclusions and Future work

In this paper we have presented a methodology to predict the sequential and parallel RTD of a given local search algorithm to tackle a given instance. In particular, we start with the prediction of the most suitable theoretical distribution to describe the behaviour of the algorithm. Based on the literature in the area, we limit our attention to the shifted exponential and lognormal distributions. Then, we use a ML regression model to learn the parameters of the distribution, and finally we apply order statistics to infer the RTD of the parallel algorithm. In this paper, we go beyond existing work aiming at predicting

individual metrics such as mean or median runtime execution, and predict the whole RTD of the algorithm. We would like to recall that the RTD fully describes the performance of randomised algorithms, and therefore, we are able to infer accurate knowledge of the behaviour of the sequential and parallel versions of the algorithm.

We evaluated the quality of the ML models using the expected RTDs to predict the mean runtime of SAT solvers namely Sparrow, probSAT, and g2wsat with up to 40 cores and the TSP solver LKH with up to 16 cores. Interestingly, we have observed a correlation coefficient of up to 0.85 for SAT solvers and up to 0.95 for the TSP solver for random instances, and a CC of about 0.65 for g2wsat and a CC of about 0.88 for LKH on structured instances.

Currently, the feature computation code relies heavily in the use of SAPS and GSAT. We plan to introduce similar features with our target solvers, i.e., Sparrow and probSAT, to improve the quality of the ML models. Furthermore, we plan to extend our study to optimisation problems where the optimal value is unknown, and use the ML to predict the upper and lower bounds of the solutions.

ACKNOWLEDGMENTS

The Insight Centre for Data Analytics is supported by SFI under Grant Number SFI/RC/2289.

REFERENCES

[1] B. Bloom, D. Grove, B. Herta, A. Sabharwal, H. Samulowitz, and V. Saraswat, *SAT'12*. Springer, 2012, ch. SatX10: A Scalable Plug&Play Parallel SAT Framework, pp. 463–468.

[2] A. Arbelaez, C. Truchet, and P. Codognet, "Using Sequential Runtime Distributions for the Parallel Speedup Prediction of SAT Local Search," *TPLP*, vol. 13, no. 4-5, pp. 625–639, 2013.

[3] Y. Wang and I. H. Witten, "Modeling for optimal probability prediction," in *ICML'02*, 2002, pp. 650–657.

[4] N. Eén and N. Sörensson, "An Extensible SAT-Solver," in *SAT'03*. Springer, 2003, pp. 502–518.

[5] M. Heule, M. Dufour, J. van Zwieten, and H. van Maaren, "March_eq: Implementing Additional Reasoning into an Efficient Look-Ahead SAT Solver," in *SAT'04*. Springer, 2004, pp. 345–359.

[6] A. Biere, "PicoSAT Essentials," *Journal on Satisfiability, Boolean Modeling and Computation, JSAT*, vol. 4, no. 2-4, pp. 75–97, 2008.

[7] H. H. Hoos and T. Stützle, "Local Search Algorithms for SAT: An Empirical Evaluation," *J. Autom. Reasoning*, vol. 24, no. 4, pp. 421–481, 2000.

[8] H. Fang and W. Ruml, "Complete Local Search for Propositional Satisfiability," in *AAAI'04*, July 2004, pp. 161–166.

[9] G. Audemard, J.-M. Lagniez, B. Mazure, and L. Sais, "Learning in Local Search," in *ICTAI'09*. IEEE Computer Society, 2009, pp. 417–424.

[10] K. Helsgaun, "An effective implementation of the lin-kernighan traveling salesman heuristic," *European Journal of Operational Research*, vol. 126, no. 1, pp. 106–130, 2000.

[11] D. L. Applegate, R. E. Bixby, V. Chvtal, and W. J. Cook, *The Traveling Salesman Problem: A Computational Study*. Princeton University Press, 2007.

[12] L. Xu, F. Hutter, H. H. Hoos, and K. Leyton-Brown, "Satzilla: Portfolio-based algorithm selection for sat," *Journal of Artificial Intelligence Research*, vol. 32, pp. 565–606, 2008.

[13] D. E. Knuth, "Estimating the efficiency of backtrack programs," *Mathematics of Computation*, vol. 29, no. 129, pp. 121–136, 1975.

[14] F. Hutter, L. Xu, H. H. Hoos, and K. Leyton-Brown, "Algorithm runtime prediction: Methods & evaluation," *Artif. Intell.*, vol. 206, pp. 79–111, 2014.

[15] L. Babai, "Monte-carlo algorithms in graph isomorphism testing," Université de Montréal, Research Report D.M.S. No. 79-10, 1979.

[16] M. Verhoeven and E. Aarts, "Parallel local search," *Journal of Heuristics*, vol. 1, no. 1, pp. 43–65, 1995.

[17] M. Luby, A. Sinclair, and D. Zuckerman, "Optimal speedup of las vegas algorithms," *Optimal Speedup of Las Vegas Algorithms*, vol. 47, no. 4, pp. 173–180, 1993.

[18] C. Truchet, F. Richoux, and P. Codognet, "Prediction of Parallel Speed-ups for Las Vegas Algorithms," in *ICPP'13*. IEEE Press, October 2013.

[19] C. P. Gomes, B. Selman, N. Crato, and H. A. Kautz, "Heavy-tailed phenomena in satisfiability and constraint satisfaction problems," *J. Autom. Reasoning*, vol. 24, no. 1/2, pp. 67–100, 2000.

[20] V. Vapnik, *The Nature of Statistical Learning*. New York, NY, USA: Springer Verlag, 1995.

[21] A. Balint and A. Fröhlich, "Improving Stochastic Local Search for SAT with a New Probability Distribution," in *SAT'10*. Springer, 2010, pp. 10–15.

[22] A. Balint and U. Schöning, "Choosing probability distributions for stochastic local search and the role of make versus break," in *SAT'12*. Springer, 2012, pp. 16–29.

[23] C. M. Li and W. Q. Huang, "Diversification and determinism in local search for satisfiability," in *SAT'05*, ser. LNCS, vol. 3569. St. Andrews, UK: Springer, 2005, pp. 158–172.

[24] D. Tompkins, A. Balint, and H. H. Hoos, "Captain Jack: New Variable Selection Heuristics in Local Search for SAT," in *SAT'11*, 2011, pp. 302–316.

[25] I. P. Gent, H. H. Hoos, P. Prosser, and T. Walsh, "Morphing: Combining structure and randomness," in *AAAI'99*, 1999, pp. 654–660.

[26] A. R. KhudaBukhsh, L. Xu, H. H. Hoos, and K. Leyton-Brown, "Satenstein: Automatically building local search sat solvers from components," in *IJCAI*, Pasadena, California, USA, July 2009, pp. 517–524.

[27] S. Wolfram, *The Mathematica Book, 5th edition*. Wolfram Media, 2003.

[28] F. Hutter, Y. Hamadi, H. H. Hoos, and K. Leyton-Brown, "Performance Prediction and Automated Tuning of Randomized and Parametric Algorithms," in *CP'06*, Nantes, France, Sept 2006, pp. 213–228.

[29] L. Lobjois and M. Lemaître, "Branch and bound algorithm selection by performance prediction," in *AAAI/IAAI*, 1998, pp. 353–358.

[30] P. Kilby, J. K. Slaney, S. Thiébaux, and T. Walsh, "Estimating search tree size," in *AAAI*, 2006.

[31] E. Horvitz, Y. Ruan, C. P. Gomes, B. Selman, and D. M. Chickering, "A bayesian approach to tackling hard computational problems," in *UAI'01*. Morgan Kaufmann, 2001, pp. 235–244.

[32] D. Devlin and B. O'Sullivan, "Satisfiability as a classification problem," in *19th Irish Conf. on Artificial Intelligence and Cognitive Science*, 2008.

[33] S. Haim and T. Walsh, "Restart Strategy Selection using Machine Learning Techniques," in *SAT'09*. Springer, 2009, pp. 312–325.

[34] K. Smith-Miles, J. I. van Hemert, and X. Y. Lim, "Understanding TSP difficulty by learning from evolved instances," in *LION'10*, 2010, pp. 266–280.

[35] L. Kotthoff, "Algorithm selection for combinatorial search problems: A survey," *AI Magazine*, vol. 35, no. 3, pp. 48–60, 2014.

[36] B. Hurley and B. O'Sullivan, "Statistical regimes and runtime prediction," in *IJCAI'15*, 2015, pp. 318–324.

[37] Z. Kiziltan and J. Mauro, "Service-oriented volunteer computing for massively parallel constraint solving using portfolios," in *CPAIOR'10*, 2010, pp. 246–251.

[38] D. Geschwender, F. Hutter, L. Kotthoff, Y. Malitsky, H. H. Hoos, and K. Leyton-Brown, "Algorithm configuration in the cloud: A feasibility study," in *LION 8*, 2014, pp. 41–46.

[39] B. Hurley, D. Mehta, and B. O'Sullivan, "Elastic solver: Balancing solution time and energy consumption," *CoRR*, vol. abs/1605.06940, 2016.